S&DS 685: Problem Set 1

Name: Ananya Krishna **NetID:** ark89 **Date:** 01/31/2025 **Collaborator:** Valentina Simon

1 Sufficiency of Memoryless Policies (Pages 1-2)

(a) We first recall that the expected total discounted reward under policy π is defined as

$$J(\pi) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t) \right].$$

This can be written as

$$J(\pi) = \sum_{t=0}^{\infty} \gamma^t \, \mathbb{E}_\pi \left[R(s_t, a_t) \right].$$

Since

$$\mathbb{E}_\pi \left[R(s_t, a_t) \right] = \sum_{s \in S} \sum_{a \in A} R(s, a) \, \mathbb{P}_\pi(s_t = s, a_t = a),$$

we have

$$J(\pi) = \sum_{t=0}^{\infty} \gamma^t \sum_{s \in S} \sum_{a \in A} R(s, a) \, \mathbb{P}_\pi(s_t = s, a_t = a).$$

We can rearrange to get:

$$J(\pi) = \sum_{s \in S} \sum_{a \in A} R(s, a) \left(\sum_{t=0}^{\infty} \gamma^t \, \mathbb{P}_\pi(s_t = s, a_t = a) \right).$$

We know

$$d_\pi^\mu(s, a) = \sum_{t=0}^{\infty} \gamma^t \, \mathbb{P}_\pi(s_t = s, a_t = a \mid s_0 \sim \mu).$$

So we can substitute to give:

$$\boxed{J(\pi) = \sum_{s \in S} \sum_{a \in A} R(s, a) \, d_\pi^\mu(s, a).}$$

As a special case, when the initial state is a fixed state e_s (that is, $\mu = \delta_{e_s}$), the value function is defined by

$$V_\pi(e_s) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t) \,\middle|\, s_0 = e_s \right].$$

We can expand the expectation as follows.

$$V_\pi(e_s) = \sum_{t=0}^{\infty} \gamma^t \, \mathbb{E}_\pi\Big[R(s_t, a_t) \,\Big|\, s_0 = e_s\Big].$$

$$\mathbb{E}_\pi\Big[R(s_t, a_t) \,\Big|\, s_0 = e_s\Big] = \sum_{s\in S}\sum_{a\in A} R(s,a)\, \mathbb{P}_\pi\Big(s_t = s,\, a_t = a \,\Big|\, s_0 = e_s\Big).$$

$$V_\pi(e_s) = \sum_{t=0}^{\infty} \gamma^t \sum_{s\in S}\sum_{a\in A} R(s,a)\, \mathbb{P}_\pi\Big(s_t = s,\, a_t = a \,\Big|\, s_0 = e_s\Big).$$

$$V_\pi(e_s) = \sum_{s\in S}\sum_{a\in A} R(s,a) \left(\sum_{t=0}^{\infty} \gamma^t \, \mathbb{P}_\pi\Big(s_t = s,\, a_t = a \,\Big|\, s_0 = e_s\Big)\right).$$

We know the *discounted visitation measure* (or occupancy measure) for the fixed initial state e_s is

$$d_\pi^{e_s}(s,a) = \sum_{t=0}^{\infty} \gamma^t \, \mathbb{P}_\pi\Big(s_t = s,\, a_t = a \,\Big|\, s_0 = e_s\Big).$$

Substituting gives

$$\boxed{V_\pi(e_s) = \sum_{s\in S}\sum_{a\in A} R(s,a)\, d_\pi^{e_s}(s,a), \quad \forall e_s \in S.}$$

(b) The Occupancy Measure Equivalence Theorem tells us that for any history-dependent policy $\pi \in \Pi$ and any initial distribution μ there exists a memoryless policy $\pi' \in \Pi_{ML}$ such that

$$d_\pi^\mu(s,a) = d_{\pi'}^\mu(s,a) \quad \text{for all } s \in S,\ a \in A.$$

In particular, if we consider the case where μ is the delta measure at any state s, it follows that

$$V_\pi(s) = \sum_{s'\in S}\sum_{a\in A} R(s',a)\, d_\pi^s(s',a) = \sum_{s'\in S}\sum_{a\in A} R(s',a)\, d_{\pi'}^s(s',a) = V_{\pi'}(s).$$

Since every history-dependent policy has an equivalent memoryless policy in terms of its occupancy measure (and thus its value), it follows that

$$\max_{\pi\in\Pi} V_\pi(s) \le \max_{\pi\in\Pi_{ML}} V_\pi(s).$$

However, because Π_{ML} is a subset of Π, we always have

$$\max_{\pi\in\Pi_{ML}} V_\pi(s) \le \max_{\pi\in\Pi} V_\pi(s).$$

Combining these two inequalities, we obtain

$$\boxed{\max_{\pi\in\Pi} V_\pi(s) = \max_{\pi\in\Pi_{ML}} V_\pi(s), \quad \forall s \in S.}$$

2 Contraction and Monotonicity of Bellman operators (Pages 3-5)

(a) Suppose that the operator $B : \mathcal{Q} \to \mathcal{Q}$ is γ-contractive with $\gamma \in [0, 1)$. We prove by contradiction that its fixed point is unique.

Assume that there exist two fixed points $q_1, q_2 \in \mathcal{Q}$ with

$$Bq_1 = q_1, \quad Bq_2 = q_2, \quad \text{and} \quad q_1 \neq q_2.$$

Then, by the contractivity of B,

$$\|q_1 - q_2\|_\infty = \|Bq_1 - Bq_2\|_\infty \leq \gamma \|q_1 - q_2\|_\infty.$$

Since $\gamma < 1$, subtracting $\gamma \|q_1 - q_2\|_\infty$ from both sides yields

$$(1 - \gamma)\|q_1 - q_2\|_\infty \leq 0.$$

Because the norm is nonnegative and $1 - \gamma > 0$, it follows that

$$\|q_1 - q_2\|_\infty = 0,$$

i.e. $q_1 = q_2$. This contradiction shows that the fixed point of B is unique.

(b) Let $q^* \in \mathcal{Q}$ be the unique fixed point of B and consider the sequence $\{q_k\}$ defined by

$$q_k = Bq_{k-1} \quad \text{with an arbitrary } q_0 \in \mathcal{Q}.$$

We prove by induction that

$$\|q_k - q^*\|_\infty \leq \gamma^k \|q_0 - q^*\|_\infty, \quad \forall k \geq 0.$$

Base case: For $k = 0$,

$$\|q_0 - q^*\|_\infty = \gamma^0 \|q_0 - q^*\|_\infty.$$

Inductive step: Assume the inequality holds for some $k \geq 0$. Then

$$\begin{aligned} \|q_{k+1} - q^*\|_\infty &= \|Bq_k - Bq^*\|_\infty \quad (\text{since } Bq^* = q^*) \\ &\leq \gamma \|q_k - q^*\|_\infty \quad (\text{by } \gamma\text{-contractivity}) \\ &\leq \gamma \cdot \gamma^k \|q_0 - q^*\|_\infty = \gamma^{k+1} \|q_0 - q^*\|_\infty. \end{aligned}$$

Thus, by induction, the claim holds for all $k \geq 0$.

(c) Now suppose that the sequence is generated by an approximate update:

$$q_k = Bq_{k-1} + e_k, \quad \text{with } \|e_k\|_\infty \le \varepsilon \quad \forall k \ge 1.$$

We wish to prove that

$$\|q_k - q^*\|_\infty \le \gamma^k \|q_0 - q^*\|_\infty + \frac{\varepsilon}{1-\gamma}.$$

Observe that

$$q_k - q^* = Bq_{k-1} - Bq^* + e_k.$$

Taking the ℓ_∞-norm and using the triangle inequality and contractivity, we have

$$\|q_k - q^*\|_\infty \le \|Bq_{k-1} - Bq^*\|_\infty + \|e_k\|_\infty \le \gamma \|q_{k-1} - q^*\|_\infty + \varepsilon.$$

Unrolling this recursion gives

$$\|q_k - q^*\|_\infty \le \gamma^k \|q_0 - q^*\|_\infty + \varepsilon \sum_{i=0}^{k-1} \gamma^i.$$

Since

$$\sum_{i=0}^{k-1} \gamma^i = \frac{1-\gamma^k}{1-\gamma} \le \frac{1}{1-\gamma},$$

it follows that

$$\|q_k - q^*\|_\infty \le \gamma^k \|q_0 - q^*\|_\infty + \frac{\varepsilon}{1-\gamma}.$$

(d) Consider the following Bellman operators. For a Q-function $Q : S \times A \to \mathbb{R}$, define

$$(TQ)(s,a) = R(s,a) + \gamma \, \mathbb{E}_{s' \sim P(\cdot|s,a)} \Big[\max_{a' \in A} Q(s', a') \Big],$$

and for a fixed policy π,

$$(T^\pi Q)(s,a) = R(s,a) + \gamma \, \mathbb{E}_{\substack{s' \sim P(\cdot|s,a) \\ a' \sim \pi(\cdot|s')}} \Big[Q(s', a') \Big].$$

i. γ–contractivity

For any $Q, Q' \in \mathcal{Q}$ and any $(s,a) \in S \times A$, consider first T:

$$\begin{aligned}
|(TQ)(s,a) - (TQ')(s,a)| &= \gamma \left| \mathbb{E}_{s' \sim P(\cdot|s,a)} \Big[\max_{a' \in A} Q(s', a') - \max_{a' \in A} Q'(s', a') \Big] \right| \\
&\le \gamma \, \mathbb{E}_{s' \sim P(\cdot|s,a)} \Big| \max_{a' \in A} Q(s', a') - \max_{a' \in A} Q'(s', a') \Big| \\
&\le \gamma \, \mathbb{E}_{s' \sim P(\cdot|s,a)} \Big[\max_{a' \in A} \big| Q(s', a') - Q'(s', a') \big| \Big] \\
&\le \gamma \, \|Q - Q'\|_\infty.
\end{aligned}$$

Taking the maximum over (s, a) shows that

$$\|TQ - TQ'\|_\infty \leq \gamma \|Q - Q'\|_\infty.$$

A similar argument works for T^π:

$$\begin{aligned}
|(T^\pi Q)(s,a) - (T^\pi Q')(s,a)| &= \gamma \left| \mathbb{E}_{\substack{s' \sim P(\cdot|s,a) \\ a' \sim \pi(\cdot|s')}} \Big[Q(s',a') - Q'(s',a') \Big] \right| \\
&\leq \gamma \, \mathbb{E}_{\substack{s' \sim P(\cdot|s,a) \\ a' \sim \pi(\cdot|s')}} \Big[\big| Q(s',a') - Q'(s',a') \big| \Big] \\
&\leq \gamma \, \|Q - Q'\|_\infty.
\end{aligned}$$

Taking the maximum over all $(s, a) \in S \times A$, we conclude that

$$\|T^\pi Q - T^\pi Q'\|_\infty \leq \gamma \|Q - Q'\|_\infty.$$

Thus, both T and T^π are γ-contractive.

i. Monotonicity

Suppose $Q, Q' \in \mathcal{Q}$ satisfy $Q(s,a) \geq Q'(s,a)$ for all $(s,a) \in S \times A$.

For T^π, we have

$$\begin{aligned}
(T^\pi Q)(s,a) &= R(s,a) + \gamma \, \mathbb{E}_{\substack{s' \sim P(\cdot|s,a) \\ a' \sim \pi(\cdot|s')}} [Q(s',a')] \\
&\geq R(s,a) + \gamma \, \mathbb{E}_{\substack{s' \sim P(\cdot|s,a) \\ a' \sim \pi(\cdot|s')}} [Q'(s',a')] \\
&= (T^\pi Q')(s,a).
\end{aligned}$$

Thus, T^π is monotone.

For T, note that for every s' and all a',

$$Q(s',a') \geq Q'(s',a'),$$

so that

$$\max_{a' \in A} Q(s',a') \geq \max_{a' \in A} Q'(s',a').$$

Hence,

$$\begin{aligned}
(TQ)(s,a) &= R(s,a) + \gamma \, \mathbb{E}_{s' \sim P(\cdot|s,a)} \Big[\max_{a' \in A} Q(s',a') \Big] \\
&\geq R(s,a) + \gamma \, \mathbb{E}_{s' \sim P(\cdot|s,a)} \Big[\max_{a' \in A} Q'(s',a') \Big] \\
&= (TQ')(s,a).
\end{aligned}$$

Thus, T is monotone

3 Episodic MDP (Pages 6-9)

(a) Recall that an episodic MDP is defined by

$$M = \big(S, A, \{P_h\}_{h=1}^H, \{R_h\}_{h=1}^H, H\big),$$

with the Bellman operators (acting on Q-functions) defined for each step $h \in [H]$ by

$$(T_h Q_{h+1})(s,a) = R_h(s,a) + \mathbb{E}_{s' \sim P_h(\cdot|s,a)}\Big[\max_{a' \in A} Q_{h+1}(s', a')\Big],$$

with terminal condition $Q_{H+1}(s,a) = 0$ for all (s,a).

The Value Iteration algorithm is defined as follows:

1. Initialize $Q_h^{(0)}(s,a) = 0$ for all $h \in [H]$ and all $(s,a) \in S \times A$.
2. For $k = 1, 2, \ldots, K$, compute for all $h \in [H]$:

   $$Q_h^{(k)}(s,a) = T_h Q_{h+1}^{(k)}(s,a),$$

   with $Q_{H+1}^{(k)}(s,a) = 0$.
3. Return the greedy policy $\bar{\pi} = \{\bar{\pi}_h\}_{h=1}^H$ defined by

   $$\bar{\pi}_h(s) \in \arg\max_{a \in A} Q_h^{(K)}(s,a).$$

The optimal Q-functions satisfy the Bellman optimality equations:

$$Q_h^*(s,a) = R_h(s,a) + \mathbb{E}_{s' \sim P_h(\cdot|s,a)}\Big[\max_{a' \in A} Q_{h+1}^*(s', a')\Big], \quad \text{with } Q_{H+1}^*(s,a) = 0.$$

We now show by backward induction that after one complete backward sweep (i.e. setting $k = 1$) we have

$$Q_h^{(1)}(s,a) = Q_h^*(s,a) \quad \text{for all } h \in [H].$$

Base case: For $h = H$, we have

$$Q_H^{(1)}(s,a) = T_H Q_{H+1}^{(1)}(s,a) = R_H(s,a) + \mathbb{E}_{s' \sim P_H(\cdot|s,a)}\Big[\max_{a' \in A} Q_{H+1}^{(1)}(s', a')\Big].$$

Since $Q_{H+1}^{(1)}(s,a) = 0$ for all (s,a), it follows that

$$Q_H^{(1)}(s,a) = R_H(s,a) = Q_H^*(s,a).$$

Inductive step: Assume that for some $h+1 \in \{2, \ldots, H+1\}$ we have

$$Q_{h+1}^{(1)}(s,a) = Q_{h+1}^{*}(s,a) \quad \forall (s,a).$$

Then for step h,

$$\begin{aligned} Q_h^{(1)}(s,a) &= T_h Q_{h+1}^{(1)}(s,a) \\ &= R_h(s,a) + \mathbb{E}_{s' \sim P_h(\cdot|s,a)} \Big[\max_{a' \in A} Q_{h+1}^{(1)}(s',a') \Big] \\ &= R_h(s,a) + \mathbb{E}_{s' \sim P_h(\cdot|s,a)} \Big[\max_{a' \in A} Q_{h+1}^{*}(s',a') \Big] \\ &= Q_h^{*}(s,a). \end{aligned}$$

Thus, by induction, $Q_h^{(1)} = Q_h^*$ for all $h \in [H]$. Consequently, the greedy policy based on $Q^{(1)}$ is the optimal policy $\pi^* = \{\pi_h^*\}_{h=1}^H$.

(b) In Policy Iteration for the episodic MDP, we perform the following:

1. **Initialization:** Choose an arbitrary nonstationary policy $\pi^{(0)} = \{\pi_h^{(0)}\}_{h=1}^H$.
2. **Policy Evaluation:** For the current policy $\pi^{(k)}$, compute the Q-functions recursively for $h = H, H-1, \ldots, 1$ by

   $$Q_h^{\pi^{(k)}}(s,a) = R_h(s,a) + \mathbb{E}_{s' \sim P_h(\cdot|s,a)} \Big[Q_{h+1}^{\pi^{(k)}}\big(s', \pi_{h+1}^{(k)}(s')\big) \Big],$$

   with $Q_{H+1}^{\pi^{(k)}}(s,a) = 0$.
3. **Policy Improvement:** For each h and each $s \in S$, set

   $$\pi_h^{(k+1)}(s) \in \arg\max_{a \in A} Q_h^{\pi^{(k)}}(s,a).$$

We now show by backward induction that after H iterations the policy $\pi^{(H)}$ is optimal.

Base: At the terminal step $h = H$, note that

$$Q_H^{\pi}(s,a) = R_H(s,a)$$

for any policy π (since $Q_{H+1}(s,a) = 0$). Therefore, the improvement step at $h = H$ chooses

$$\pi_H^{(1)}(s) \in \arg\max_{a \in A} R_H(s,a),$$

which is optimal at the final step.

Inductive step: Suppose that after k iterations the improved policy $\pi_{h+1}^{(k)}$ is optimal for step $h+1$; that is, for all s,

$$Q_{h+1}^{*}(s,a) = Q_{h+1}^{\pi^{(k)}}(s,a) \quad \text{and} \quad \pi_{h+1}^{(k)}(s) \in \arg\max_{a \in A} Q_{h+1}^{*}(s,a).$$

Now consider the policy evaluation at step h:

$$Q_h^{\pi^{(k)}}(s,a) = R_h(s,a) + \mathbb{E}_{s' \sim P_h(\cdot|s,a)}\Big[Q_{h+1}^{\pi^{(k)}}\big(s', \pi_{h+1}^{(k)}(s')\big)\Big].$$

Since by the induction hypothesis $Q_{h+1}^{\pi^{(k)}} = Q_{h+1}^*$, it follows that

$$Q_h^{\pi^{(k)}}(s,a) = R_h(s,a) + \mathbb{E}_{s' \sim P_h(\cdot|s,a)}\Big[\max_{a' \in A} Q_{h+1}^*(s',a')\Big] = Q_h^*(s,a).$$

Thus, the policy improvement step at time h will set

$$\pi_h^{(k+1)}(s) \in \arg\max_{a \in A} Q_h^*(s,a),$$

i.e. $\pi_h^{(k+1)}$ is optimal. Since the horizon is H, after H iterations we have $\pi^{(H)} = \pi^*$.

(c) Let the discounted MDP be

$$M = \big(S, A, P, R, p_0, \gamma\big),$$

with discount factor $\gamma \in [0,1)$. For any $H > 0$ we define an episodic MDP

$$\hat{M} = \big(S, A, \{\hat{P}_h\}_{h=1}^H, \{\hat{R}_h\}_{h=1}^H, p_0, H\big)$$

by setting, for all $h \in [H]$:

$$\hat{P}_h = P \quad \text{and} \quad \hat{R}_h(s,a) = \gamma^{\,h-1} R(s,a).$$

Let $\hat{\pi} = \{\hat{\pi}_h\}_{h=1}^H$ be the optimal policy for $\hat{M}$. Based on $\hat{\pi}$ we define a new nonstationary policy $\bar{\pi} = \{\bar{\pi}_t\}_{t \geq 0}$ for the original MDP M as follows:

- For $t = 0, 1, \ldots, H-1$, set $\bar{\pi}_t = \hat{\pi}_{t+1}$.
- For $t \geq H$, set $\bar{\pi}_t(s) = \text{Uniform}(A)$ for all $s \in S$.

We wish to show that on M the value loss satisfies

$$\|V^* - V^{\bar{\pi}}\|_\infty \leq \frac{\gamma^H}{1-\gamma}.$$

Proof: Consider any state $s \in S$. Under the policy $\bar{\pi}$, the first H steps are taken according to $\hat{\pi}$ (which is optimal for $\hat{M}$), and thereafter actions are arbitrary. Thus, we can decompose the value functions as follows:

$$V^*(s) = \max_\pi \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t) \,\middle|\, s_0 = s\right]$$

and

$$V^{\bar{\pi}}(s) = \mathbb{E}_{\bar{\pi}}\left[\sum_{t=0}^{H-1} \gamma^t R(s_t, a_t) + \sum_{t=H}^{\infty} \gamma^t R(s_t, a_t) \,\middle|\, s_0 = s\right].$$

Since $\hat{\pi}$ is optimal for the episodic MDP $\hat{M}$, the first H steps under $\bar{\pi}$ achieve the maximal *partial* return:

$$\mathbb{E}_{\bar{\pi}}\left[\sum_{t=0}^{H-1} \gamma^t R(s_t, a_t) \,\middle|\, s_0 = s\right] = \max_{\pi'} \mathbb{E}_{\pi'}\left[\sum_{t=0}^{H-1} \gamma^t R(s_t, a_t) \,\middle|\, s_0 = s\right].$$

Thus, the only gap between $V^*(s)$ and $V^{\bar{\pi}}(s)$ is due to the remaining tail (from $t = H$ onward). In the worst case, the maximum cumulative reward starting at time H (assuming rewards are bounded by 1) is

$$\sum_{t=H}^{\infty} \gamma^t = \frac{\gamma^H}{1-\gamma}.$$

Therefore, for all $s \in S$,

$$V^*(s) - V^{\bar{\pi}}(s) \leq \frac{\gamma^H}{1-\gamma},$$

and hence

$$\boxed{\|V^* - V^{\bar{\pi}}\|_\infty \leq \frac{\gamma^H}{1-\gamma}.}$$

4 Exponential Update and Mirror Descent (Pages 10-13)

In the bandit setting we wish to solve

$$\max_{\pi \in \Delta(A)} f(\pi) = \langle R, \pi \rangle,$$

via the mirror descent update that at the current iterate π_0 computes

$$\min_{\pi \in \Delta(A)} \left\{ -\alpha \langle R, \pi \rangle + \mathrm{KL}(\pi \parallel \pi_0) \right\}.$$

Defining

$$F(\pi) = \mathrm{KL}(\pi \parallel \pi_0) - \alpha \langle R, \pi \rangle,$$

we now answer the questions.

(a) The optimization problem is

$$\min_{\pi \in \Delta(A)} F(\pi) \quad \text{where} \quad \Delta(A) = \Big\{ \pi \in \mathbb{R}^m : \ \pi(a) \geq 0, \ \sum_{a \in A} \pi(a) = 1 \Big\}.$$

To enforce the constraints we introduce Lagrange multipliers:

- $\lambda = (\lambda_1, \ldots, \lambda_m) \in \mathbb{R}^m$ for the inequality constraints $\pi(a_i) \geq 0$ (with $\lambda_i \geq 0$ for each i),
- $\nu \in \mathbb{R}$ for the equality constraint $\sum_{i=1}^m \pi(a_i) = 1$.

The Lagrangian is then

$$L(\pi, \lambda, \nu) = F(\pi) - \sum_{i=1}^m \lambda_i \, \pi(a_i) + \nu \left(\sum_{i=1}^m \pi(a_i) - 1 \right).$$

Recall that the KL divergence is given by

$$\mathrm{KL}(\pi \parallel \pi_0) = \sum_{i=1}^m \pi(a_i) \ln \frac{\pi(a_i)}{\pi_0(a_i)}.$$

Thus, writing out $F(\pi)$ we have

$$L(\pi, \lambda, \nu) = \sum_{i=1}^m \pi(a_i) \ln \frac{\pi(a_i)}{\pi_0(a_i)} - \alpha \sum_{i=1}^m R(a_i) \pi(a_i) - \sum_{i=1}^m \lambda_i \, \pi(a_i) + \nu \left(\sum_{i=1}^m \pi(a_i) - 1 \right).$$

For any fixed λ, ν we now minimize $L(\pi, \lambda, \nu)$ with respect to $\pi \geq 0$. Since the Lagrangian is separable in the coordinates $\pi(a_i)$, we differentiate with respect to $\pi(a_i)$. For each i, note that

$$\frac{d}{d\pi(a_i)} \Big[\pi(a_i) \ln \frac{\pi(a_i)}{\pi_0(a_i)} \Big] = \ln \frac{\pi(a_i)}{\pi_0(a_i)} + 1.$$

Thus, setting the derivative with respect to $\pi(a_i)$ to zero yields:

$$\ln \frac{\pi(a_i)}{\pi_0(a_i)} + 1 - \alpha R(a_i) - \lambda_i + \nu = 0.$$

Equivalently,

$$\ln \frac{\pi(a_i)}{\pi_0(a_i)} = \alpha R(a_i) + \lambda_i - \nu - 1.$$

Exponentiating both sides gives

$$\frac{\pi(a_i)}{\pi_0(a_i)} = \exp\Big\{\alpha R(a_i) + \lambda_i - \nu - 1\Big\},$$

so that the optimal solution (for fixed λ, ν) is

$$\pi^\star(a_i) = \pi_0(a_i) \exp\Big\{\alpha R(a_i) + \lambda_i - \nu - 1\Big\}, \quad \forall\, i = 1, \ldots, m.$$

It is convenient to note that the constant term $-\nu - 1$ can be lumped together and later determined by the constraint $\sum_{i=1}^m \pi^\star(a_i) = 1$.

(b) The Lagrange dual function is defined as

$$g(\lambda, \nu) = \min_{\pi \geq 0} L(\pi, \lambda, \nu) = L\big(\pi^\star, \lambda, \nu\big).$$

Substitute the expression for $\pi^\star(a_i)$ into L. Observe that by construction

$$\ln \frac{\pi^\star(a_i)}{\pi_0(a_i)} = \alpha R(a_i) + \lambda_i - \nu - 1.$$

Thus, we have

$$\begin{aligned}
L(\pi^\star, \lambda, \nu) &= \sum_{i=1}^m \pi^\star(a_i)\Big(\alpha R(a_i) + \lambda_i - \nu - 1\Big) - \alpha \sum_{i=1}^m R(a_i)\pi^\star(a_i) - \sum_{i=1}^m \lambda_i\, \pi^\star(a_i) + \nu\left(\sum_{i=1}^m \pi^\star(a_i) - 1\right) \\
&= \sum_{i=1}^m \pi^\star(a_i)\Big[\alpha R(a_i) + \lambda_i - \nu - 1 - \alpha R(a_i) - \lambda_i + \nu\Big] - \nu \\
&= -\sum_{i=1}^m \pi^\star(a_i) - \nu \\
&= - = \sum_{i=1}^m \pi_0(a_i) \exp\Big\{\alpha R(a_i) + \lambda_i - \nu - 1\Big\} - \nu
\end{aligned}$$

Thus, the dual function is

$$g(\lambda, \nu) = \sum_{i=1}^{m} \pi_0(a_i) \exp\Big\{\alpha R(a_i) + \lambda_i - \nu - 1\Big\} - \nu,$$

Hence, in the dual problem

$$\max_{\lambda \geq 0,\ \nu \in \mathbb{R}} g(\lambda, \nu),$$

As λ increases, (g(λ,ν)) decreases; Therefore (g(λ, ν)) is maximized when $\lambda^\star = 0$.

This is the answer for part (b).

(c) Now, setting $\lambda^\star = 0$ in the expression from part (a), we have the candidate optimal solution:

$$\pi^\star(a) = \pi_0(a) \exp\Big\{\alpha R(a) - \nu - 1\Big\}, \quad \forall\, a \in A.$$

To ensure that $\pi^\star$ is a valid probability distribution, we must have

$$\sum_{a \in A} \pi^\star(a) = 1.$$

That is,

$$\sum_{a \in A} \pi_0(a) \exp\Big\{\alpha R(a) - \nu - 1\Big\} = 1.$$

Factor out the constant $\exp\{-\nu - 1\}$:

$$\exp\{-\nu - 1\} \sum_{a \in A} \pi_0(a) \exp\{\alpha R(a)\} = 1.$$

Thus,

$$\exp\{-\nu - 1\} = \frac{1}{\sum_{a \in A} \pi_0(a) \exp\{\alpha R(a)\}}.$$

Taking logarithms of both sides yields

$$-\nu - 1 = -\ln\Big(\sum_{a \in A} \pi_0(a) \exp\{\alpha R(a)\}\Big),$$

so that

$$\boxed{\nu^\star = \ln\Big(\sum_{a \in A} \pi_0(a) \exp\{\alpha R(a)\}\Big) - 1.}$$

Denote this value by $\nu^\star$. Then the candidate solution becomes

$$\pi^\star(a) = \pi_0(a) \exp\Big\{\alpha R(a) - \nu^\star - 1\Big\} = \pi_0(a) \exp\Big\{\alpha R(a) - \Big(\ln\Big(\sum_{a' \in A} \pi_0(a') \exp\{\alpha R(a')\}\Big) - 1\Big) - 1\Big\}.$$

A short calculation shows that

$$\pi^\star(a) = \frac{\pi_0(a)\exp\{\alpha R(a)\}}{\sum_{a' \in A} \pi_0(a')\exp\{\alpha R(a')\}}.$$

This is exactly the desired closed form (2). One may verify that with $\lambda^\star = 0$ and $\nu^\star$ as above the KKT conditions (stationarity, primal/dual feasibility, and complementary slackness) are satisfied.

Thus, we conclude that the mirror descent update (1) yields the exponential update

$$\boxed{\pi_{\text{new}}(a) = \frac{\pi_0(a)\exp\{\alpha R(a)\}}{\sum_{a' \in A} \pi_0(a')\exp\{\alpha R(a')\}}, \quad \forall\, a \in A,}$$

which is the optimal solution to the update problem.

5 Entropy-Regularized MDP (Pages 14-16)

(a) We know that the solution of the entropy-regularized reward maximization problem is the *softmax policy*

$$\pi^*(a|s) = \frac{\exp(R(s,a)/\beta)}{\sum_a \exp(R(s,a')/\beta)}.$$

For any a, we know by log rules that

$$\log \pi^*(a|s) = \frac{R(s,a)}{\beta} - \log\left(\sum_{a'} \exp\big(R(s,a')/\beta\big)\right).$$

Thus the entropy under π^* is

$$\begin{aligned} H\big(\pi^*(\cdot|s)\big) &= -\sum_a \pi^*(a|s) \log \pi^*(a|s) \\ &= -\sum_a \pi^*(a|s) \left(\frac{R(s,a)}{\beta} - \log\left(\sum_{a'} \exp\big(R(s,a')/\beta\big)\right)\right) \\ &= -\frac{1}{\beta}\sum_a \pi^*(a|s) R(s,a) + \log\left(\sum_{a'} \exp\big(R(s,a')/\beta\big)\right). \end{aligned}$$

Plugging this back into the objective gives

$$\begin{aligned} F^*(s) &= \sum_a \pi^*(a|s) R(s,a) + \beta\, H\big(\pi^*(\cdot|s)\big) \\ &= \sum_a \pi^*(a|s) R(s,a) + \beta \left[-\frac{1}{\beta}\sum_a \pi^*(a|s) R(s,a) + \log\left(\sum_{a'} \exp\big(R(s,a')/\beta\big)\right)\right] \\ &= \beta\, \log\left(\sum_a \exp(R(s,a)/\beta)\right). \end{aligned}$$

Thus, we have shown that

$$\boxed{F^*(s) = \beta\, \log\left(\sum_a \exp\left(\frac{R(s,a)}{\beta}\right)\right).}$$

(b) We wish to prove that

$$0 \le F^*(s) - \max_a R(s,a) \le \beta \log |A|,$$

where $|A|$ is the number of actions.

Lower Bound: $F^*(s) - \max_a R(s,a) \ge 0$

Consider a deterministic (greedy) policy $\pi_{\text{greedy}}(\cdot|s)$ that puts all its probability mass on some action a^* with $R(s,a^*) = \max_a R(s,a)$. For any deterministic policy the entropy is zero, and hence the objective becomes

$$\sum_a \pi_{\text{greedy}}(a|s)R(s,a) + \beta\, H\big(\pi_{\text{greedy}}(\cdot|s)\big) = \max_a R(s,a).$$

Since $F^*(s)$ is the maximum over all policies, it follows that

$$F^*(s) \geq \max_a R(s,a),$$

or equivalently,

$$F^*(s) - \max_a R(s,a) \geq 0.$$

Upper Bound: $F^*(s) - \max_a R(s,a) \leq \beta \log |A|$

Recall that we have

$$F^*(s) = \beta\, \log \left(\sum_a \exp(R(s,a)/\beta) \right).$$

We can factor out $\max_a R(s,a)$ as follows:

$$\begin{aligned} F^*(s) &= \beta\, \log \left(\sum_a \exp\left(\frac{R(s,a)}{\beta} \right) \right) \\ &= \beta\, \log \left(\exp\left(\frac{\max_a R(s,a)}{\beta} \right) \sum_a \exp\left(\frac{R(s,a) - \max_a R(s,a)}{\beta} \right) \right) \\ &= \max_a R(s,a) + \beta\, \log \left(\sum_a \exp\left(\frac{R(s,a) - \max_a R(s,a)}{\beta} \right) \right). \end{aligned}$$

Since $R(s,a) - \max_a R(s,a) \leq 0$ for all a, we have

$$\exp\left(\frac{R(s,a) - \max_a R(s,a)}{\beta} \right) \leq 1.$$

Thus,

$$\sum_a \exp\left(\frac{R(s,a) - \max_a R(s,a)}{\beta} \right) \leq |A|,$$

and so

$$F^*(s) - \max_a R(s,a) \leq \beta\, \log |A|.$$

Therefore

$$\boxed{0 \leq F^*(s) - \max_a R(s,a) \leq \beta\, \log |A|.}$$

(c) Now consider the following entropy-regularized problem for a given Q-function:

$$\max_{\pi(\cdot|s)} \sum_a \pi(a|s)Q(s,a) + \beta\, H\big(\pi(\cdot|s)\big).$$

By an argument analogous to that in part (a), one can show that the optimal policy is

$$\pi_{\text{soft}}(a|s) = \frac{\exp\left(\frac{Q(s,a)}{\beta}\right)}{\sum_{a' \in A} \exp\left(\frac{Q(s,a')}{\beta}\right)}.$$

Furthermore, the corresponding optimal value is

$$\begin{aligned} V_{\text{soft}}(s) &= \sum_{a \in A} \pi_{\text{soft}}(a|s)Q(s,a) + \beta\, H\big(\pi_{\text{soft}}(\cdot|s)\big) \\ &= \beta\, \log\left(\sum_{a \in A} \exp\left(\frac{Q(s,a)}{\beta}\right)\right). \end{aligned}$$

Thus, we have

$$\boxed{\begin{aligned} \pi_{\text{soft}}(a|s) &= \frac{\exp\left(Q(s,a)/\beta\right)}{\sum_{a' \in A} \exp\left(Q(s,a')/\beta\right)}, \\ V_{\text{soft}}(s) &= \beta\, \log\left(\sum_{a \in A} \exp\left(Q(s,a)/\beta\right)\right). \end{aligned}}$$

This shows that the softmax operator is the *entropy-regularized* counterpart of the hard maximum (or greedy) operator.

Train a PPO agent on various environments

Your task is to configure and initialize a Proximal Policy Optimization (PPO) agent using the Stable Baselines3 (SB3) library to train it on various environment from Gymnasium. You will be responsible for setting key hyperparameters that define the agent's learning behavior and neural network architecture. Understanding and appropriately configuring these hyperparameters is crucial for effective training and achieving optimal performance.

Your Tasks: You need to complete

1. **Configure the Neural Network Architecture (`ppo_network_args`):**
    - **Hidden Layers:**
        - Determine and set the number of neurons in each hidden layer.
        - Adjust the number of hidden layers if necessary.
    - **Activation Function:**
        - Ensure the activation function is appropriately set (already set to `ReLU` in the snippet).
2. **Set Hyperparameters in `model = PPO( )` function:**
    - **Learning Rate (`learning_rate`):**
        - Decide on an appropriate learning rate that balances convergence speed and stability.
    - **Batch Size (`batch_size`):**
        - Choose a batch size that efficiently utilizes computational resources without causing memory issues.
    - **Number of Steps (`n_steps`):**
        - Set the number of steps the agent takes in each environment before updating the policy.
    - **Policy Keyword Arguments (`policy_kwargs`):**
        - Integrate the defined network architecture into the PPO agent.

Hyperparameter Definitions and Guidance:

To successfully complete this task, it's essential to understand the role of each hyperparameter in the PPO algorithm and how it influences the agent's learning process.

1. **Neural Network Architecture (`ppo_network_args`):**

- **`net_arch`:**
  - **Definition:** Specifies the architecture of the neural network used for the policy and value functions.
  - **Purpose:** Determines the complexity and capacity of the model to learn intricate patterns in the environment.
  - **Example:** `[64, 64]` implies two hidden layers, each with 64 neurons.
- **`activation_fn`:**
  - **Definition:** Activation function applied after each hidden layer.
  - **Purpose:** Introduces non-linearity, allowing the network to learn complex relationships.
  - **Common Choices:** `ReLU`, `Tanh`, `Sigmoid`.
  - **Current Setting:** `torch.nn.ReLU` (Rectified Linear Unit).

2. **Learning Rate (`learning_rate`):**
   - **Definition:** The step size at each iteration while moving toward a minimum of the loss function.
   - **Purpose:** Controls how much the model is updated during training.
   - **Impact:**
     - **Too High:** Can cause the model to converge too quickly to a suboptimal solution or diverge.
     - **Too Low:** May result in very slow convergence, requiring more training steps.
   - **Typical Range:** `1e-5` to `1e-3`.
   - **Current Setting:** `4e-4` (0.0004), a common default value.
3. **Batch Size (`batch_size`):**
   - **Definition:** The number of samples processed before the model is updated.
   - **Purpose:** Balances the trade-off between computational efficiency and the stability of the learning process.
   - **Impact:**
     - **Large Batch Size:** Provides more accurate gradient estimates but requires more memory and computational resources.
     - **Small Batch Size:** Reduces memory usage and can lead to faster iterations but may result in noisier gradient estimates.
   - **Typical Range:** `32` to `512`.
   - **Current Setting:** `128`, a standard mid-range value.
4. **Number of Steps (`n_steps`):**

- **Definition:** Number of steps to run for each environment per update.
- **Purpose:** Determines how much experience the agent gathers before performing a policy update.
- **Impact:**
  - **High `n_steps`:** More data per update can lead to more stable and informed updates but may require more memory.
  - **Low `n_steps`:** More frequent updates with less data, which can make learning faster but potentially less stable.
- **Typical Range:** `128` to `2048`.
- **Current Setting:** `1024`, balancing between stability and efficiency.

5. **Device (`device`):**

- **Definition:** Specifies whether to run the model on CPU or GPU.
- **Purpose:** Utilizes available hardware accelerators to speed up training.
- **Impact:**
  - **CPU:** Slower training but universally compatible.
  - **GPU:** Faster training for large models and datasets but requires compatible hardware.
- **Current Setting:** `device`, typically set to `"auto"` to let SB3 decide based on availability.

In [1]:
```
import os
import base64
import numpy as np
import matplotlib.pyplot as plt
from pathlib import Path
import gymnasium as gym
from stable_baselines3 import PPO
from stable_baselines3.common.vec_env import VecVideoRecorder, DummyVecEnv
from stable_baselines3.common.env_util import make_vec_env
from stable_baselines3.common.callbacks import BaseCallback
import torch
import random
from io import BytesIO
import imageio
import warnings
from IPython.display import Image, display

warnings.filterwarnings('ignore')



# GPU/CPU detection
device = "cuda" if torch.cuda.is_available() else "cpu"
```

```
print("="*50)
print(f"Using device: {device.upper()}")
if device == "cuda":
    print(f"GPU: {torch.cuda.get_device_name(0)}")
print("="*50)
```

```
==================================================
Using device: CUDA
GPU: Tesla V100-SXM2-32GB
==================================================
```

Helper functions that evaluates the policy and plots a random evaluated trajectory in Gif

In [6]:
```
def generate_gif(frames, fps=30):
    """
    Generates an animated GIF from a list of frames with infinite looping.

    Args:
        frames (list): List of frames (as NumPy arrays).
        fps (int): Frames per second for the GIF.

    Returns:
        gif_bytes (bytes): The GIF image in bytes.
    """
    with BytesIO() as buffer:
        # 'loop=0' ensures the GIF loops infinitely
        imageio.mimsave(buffer, frames, format='GIF', fps=fps, loop=0)
        gif_bytes = buffer.getvalue()
    return gif_bytes

def display_gif(gif_bytes):
    """
    Displays an animated GIF in the Jupyter notebook with infinite looping.

    Args:
        gif_bytes (bytes): The GIF image in bytes.
    """
    display(Image(data=gif_bytes))

def evaluate_policy_sb3(model, env, num_episodes=10, visualize_every=1, max_
    """
    Evaluates a Stable Baselines3 PPO model by running multiple episodes, ca
    generating a GIF for a randomly selected episode, and printing average r

    Args:
        model (stable_baselines3.PPO): The trained PPO model.
        env (gym.Env): The Gymnasium environment.
```

```
        num_episodes (int): Number of episodes to evaluate.
        visualize_every (int): Frequency of episodes to visualize (e.g., eve
        max_steps (int): Maximum number of steps per episode.

    Returns:
        avg_reward (float): The average reward over all episodes.
    """
    returns = []
    all_frames = []  # To store frames of all episodes

    for episode in range(1, num_episodes + 1):
        frames = []
        # Seed the environment differently for each episode for variability
        state, _ = env.reset(seed=256 * episode)
        done = False
        total_reward = 0
        step = 0

        while not done and step < max_steps:
            # SB3 models expect the state in the environment's observation s
            # The .predict() method returns the action and the state (for re
            action, _ = model.predict(state, deterministic=True)
            state, reward, done, truncated, info = env.step(action)
            total_reward += reward

            # Render and collect frame
            frame = env.render()
            if frame is not None:
                frames.append(frame)
            step += 1

        returns.append(total_reward)
        all_frames.append(frames)  # Store frames for this episode
        print(f"Episode {episode}: Total Reward: {total_reward:.2f}")

    # Compute average reward
    avg_reward = np.mean(returns)
    print(f"\n**Average Reward over {num_episodes} episodes:** {avg_reward:.

    # Randomly select one episode's frames to generate and display GIF
    # selected_episode = random.randint(0, num_episodes - 1)
    # selected_frames = all_frames[selected_episode]
    # print(f"\n**Displaying GIF for Randomly Selected Episode {selected_epi
    # gif_bytes = generate_gif(selected_frames)
    # display_gif(gif_bytes)

    return avg_reward
```

Log the reward information during training

In [3]:
```python
from stable_baselines3.common.callbacks import BaseCallback
import numpy as np

class TrainingLogger(BaseCallback):
    def __init__(self, log_interval=5000):
        super().__init__()
        self.log_interval = log_interval
        self.episode_rewards = []
        self.last_log_step = 0

    def _on_step(self) -> bool:
        # Log every log_interval steps across all environments
        if (self.num_timesteps - self.last_log_step) >= self.log_interval:
            self.last_log_step = self.num_timesteps
            if len(self.model.ep_info_buffer) > 0:
                mean_reward = np.mean([ep_info["r"] for ep_info in self.mode
                self.episode_rewards.append(mean_reward)
                print(f"Step {self.num_timesteps}:")
                print(f"  Average reward (last {len(self.model.ep_info_buffe
                print(f"  Current learning rate: {self.model.learning_rate:.
        return True
```

Train a PPO Agent on LunarLander-v3

Hyperparameters:

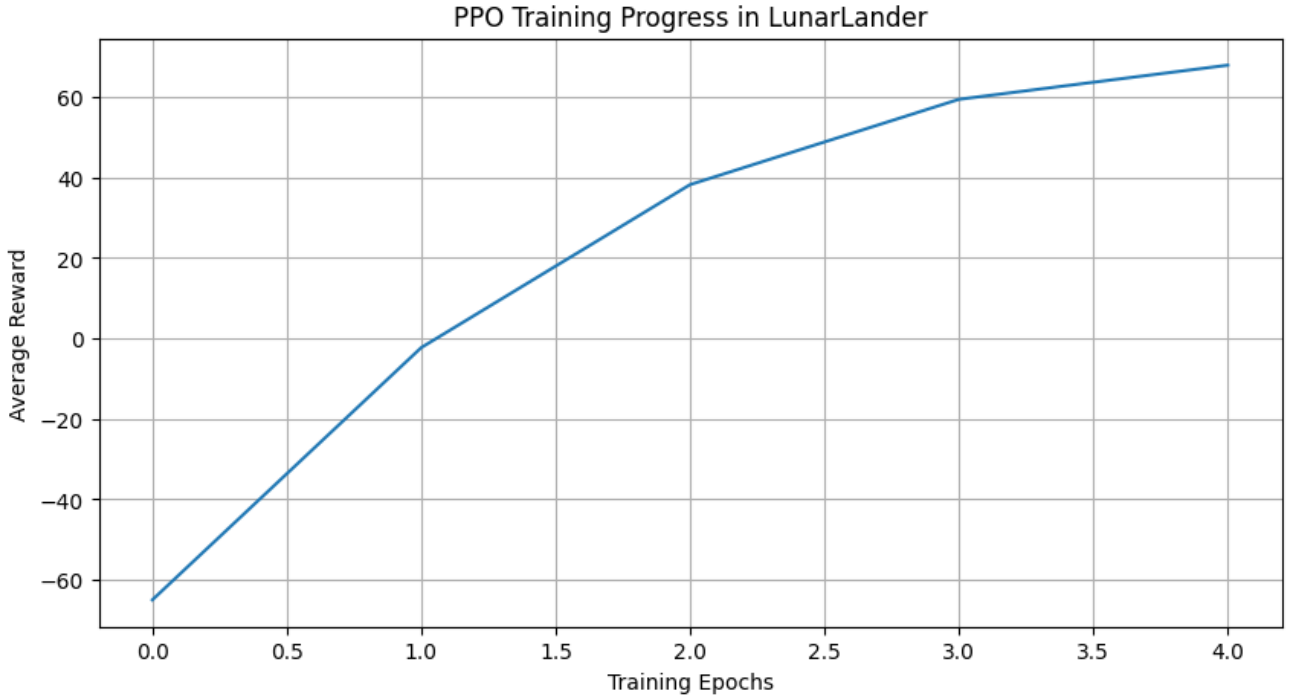
- **Hidden Layer Dimensions (`net_arch`):** Use two hidden layers with `64` neurons each.
- **Activation Function (`activation_fn`):** Apply the ReLU activation function.
- **Learning Rate (`learning_rate`):** Set the learning rate to `4e-4` .
- **Batch Size (`batch_size`):** Configure the batch size to `128` .
- **Number of Steps (`n_steps`):** Set the number of steps per update to `1024` .
- **Device (`device`):** Set the device to `"auto"` to utilize available hardware accelerators.

.

In [4]:
```python
# %% Revised PPO Training Cell with Less Logging

##########################################################
##### START OF CODE TO BE PRINTED  ###################
##########################################################

ENV_ID = "LunarLander-v3"
```

```
env = make_vec_env(ENV_ID, n_envs=4)

ppo_network_args = dict(
    net_arch=dict(pi=[64, 64], vf=[64, 64]),
    activation_fn=torch.nn.ReLU
)

model = PPO(
    "MlpPolicy",
    env,
    learning_rate=4e-4,
    batch_size=128,
    n_steps=1024,
    policy_kwargs=ppo_network_args,
    device="auto"
)

print("\nNeural Network Architecture:")
print("Actor Network (Policy): Input -> 64 -> ReLU -> 64 -> ReLU -> Output")
print("Critic Network (Value): Input -> 64 -> ReLU -> 64 -> ReLU -> Output\r


#########################################################
##### END OF CODE TO BE PRINTED ##################
#########################################################


# Training
logger = TrainingLogger(log_interval=40000)
model.learn(total_timesteps=200000, callback=logger)


# Plot training progress
plt.figure(figsize=(10, 5))
plt.plot(logger.episode_rewards)
plt.title("PPO Training Progress in LunarLander")
plt.xlabel("Training Epochs")
plt.ylabel("Average Reward")
plt.grid()
plt.show()
```

```
Neural Network Architecture:
Actor Network (Policy): Input -> 64 -> ReLU -> 64 -> ReLU -> Output
Critic Network (Value): Input -> 64 -> ReLU -> 64 -> ReLU -> Output

Step 40000:
  Average reward (last 100 episodes): -65.01
  Current learning rate: 4.00e-04
Step 80000:
  Average reward (last 100 episodes): -2.27
  Current learning rate: 4.00e-04
Step 120000:
  Average reward (last 100 episodes): 38.27
  Current learning rate: 4.00e-04
Step 160000:
  Average reward (last 100 episodes): 59.49
  Current learning rate: 4.00e-04
Step 200000:
  Average reward (last 100 episodes): 67.98
  Current learning rate: 4.00e-04
```



In [5]:
```python
# Evaluate the policy
print(f"Evaluating policy on {ENV_ID}")
# Initialize Gymnasium environment
env = gym.make(ENV_ID, render_mode="rgb_array")

# Evaluate the trained policy
num_eval_episodes = 5
visualize_every = 1  # Visualize every episode
avg_reward = evaluate_policy_sb3(model, env, num_episodes=num_eval_episodes,
```

```
env.close()
```

```
Evaluating policy on LunarLander-v3
Episode 1: Total Reward: 140.76
Episode 2: Total Reward: -48.42
Episode 3: Total Reward: 145.89
Episode 4: Total Reward: 222.88
Episode 5: Total Reward: 207.56

**Average Reward over 5 episodes:** 133.73

**Displaying GIF for Randomly Selected Episode 2:**
<IPython.core.display.Image object>
```

Train a PPO Agent on BipedalWalker Environment

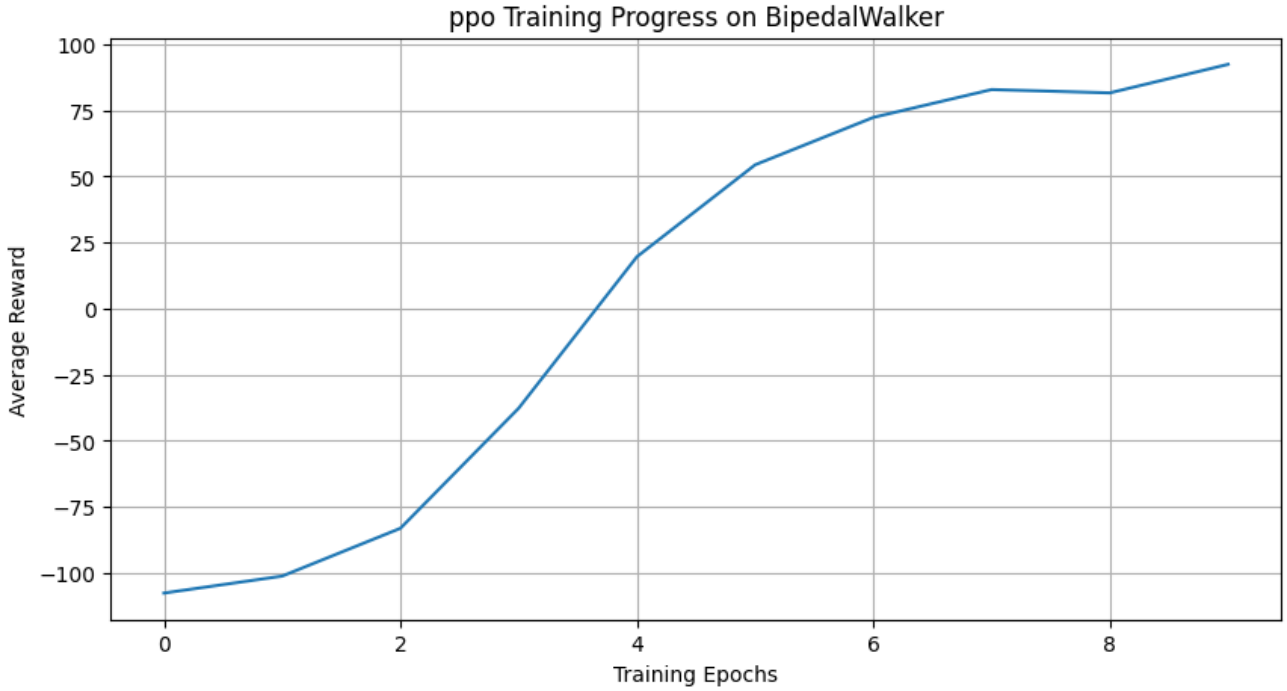
Hyperparameters:

- **Hidden Layer Dimensions (`net_arch`):** Use two hidden layers with `256` neurons each.
- **Activation Function (`activation_fn`):** Apply the ReLU activation function.
- **Learning Rate (`learning_rate`):** Set the learning rate to `1e-4` .
- **Batch Size (`batch_size`):** Configure the batch size to `128` .
- **Number of Steps (`n_steps`):** Set the number of steps per update to `1024` .
- **Device (`device`):** Set the device to `"auto"` to utilize available hardware accelerators.

In [7]:
```
# %% Revised PPO Training Cell with Less Logging

########################################################
##### START OF CODE TO BE PRINTED  ###################
########################################################

ENV_ID = "BipedalWalker-v3"

env = make_vec_env(ENV_ID, n_envs=4)

ppo_network_args = ppo_network_args = dict(
    net_arch=dict(pi=[256, 256], vf=[256, 256]),
    activation_fn=torch.nn.ReLU
)

model = PPO(
    "MlpPolicy",
    env,
    learning_rate=1e-4,
```

```
    batch_size=128,
    n_steps=1024,
    policy_kwargs=ppo_network_args,
    device="auto"
)


print("\nNeural Network Architecture:")
print("Actor Network (Policy): Input -> 256 -> ReLU -> 256 -> ReLU -> Output
print("Critic Network (Value): Input -> 256 -> ReLU -> 256 -> ReLU -> Output


########################################################
##### END OF CODE TO BE PRINTED ####################
########################################################









# Training
logger = TrainingLogger(log_interval=50000)
model.learn(total_timesteps=500000, callback=logger)


# Plot training progress
plt.figure(figsize=(10, 5))
plt.plot(logger.episode_rewards)
plt.title("ppo Training Progress on BipedalWalker")
plt.xlabel("Training Epochs")
plt.ylabel("Average Reward")
plt.grid()
plt.show()
```

```
Neural Network Architecture:
Actor Network (Policy): Input -> 256 -> ReLU -> 256 -> ReLU -> Output
Critic Network (Value): Input -> 256 -> ReLU -> 256 -> ReLU -> Output

Step 50000:
  Average reward (last 71 episodes): -107.60
  Current learning rate: 1.00e-04
Step 100000:
  Average reward (last 100 episodes): -101.18
  Current learning rate: 1.00e-04
Step 150000:
  Average reward (last 100 episodes): -83.10
  Current learning rate: 1.00e-04
Step 200000:
  Average reward (last 100 episodes): -37.69
  Current learning rate: 1.00e-04
Step 250000:
  Average reward (last 100 episodes): 19.62
  Current learning rate: 1.00e-04
Step 300000:
  Average reward (last 100 episodes): 54.41
  Current learning rate: 1.00e-04
Step 350000:
  Average reward (last 100 episodes): 72.34
  Current learning rate: 1.00e-04
Step 400000:
  Average reward (last 100 episodes): 82.88
  Current learning rate: 1.00e-04
Step 450000:
  Average reward (last 100 episodes): 81.62
  Current learning rate: 1.00e-04
Step 500000:
  Average reward (last 100 episodes): 92.41
  Current learning rate: 1.00e-04
```



In [8]:
```
# Evaluate the policy
print(f"Evaluating policy on {ENV_ID}")
# Initialize Gymnasium environment
env = gym.make(ENV_ID, render_mode="rgb_array")

# Evaluate the trained policy
num_eval_episodes = 5
visualize_every = 1  # Visualize every episode
avg_reward = evaluate_policy_sb3(model, env, num_episodes=num_eval_episodes,

env.close()
```

```
Evaluating policy on BipedalWalker-v3
Episode 1: Total Reward: -106.80
Episode 2: Total Reward: -110.09
Episode 3: Total Reward: -106.60
Episode 4: Total Reward: -110.00
Episode 5: Total Reward: -104.53

**Average Reward over 5 episodes:** -107.60
```

Train a PPO Agent on Ant-v5 environment

Hyperparameters:

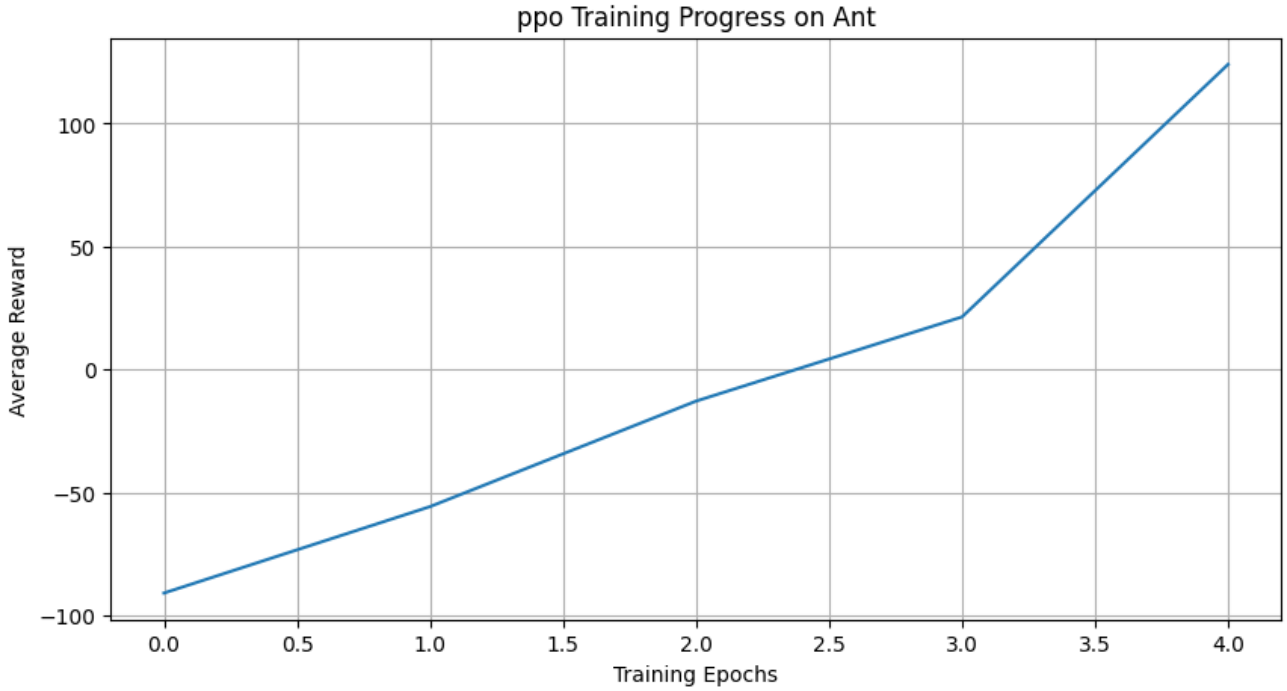
- **Hidden Layer Dimensions (`net_arch`):** Use two hidden layers with `512` neurons each.
- **Activation Function (`activation_fn`):** Apply the ReLU activation function.

- **Learning Rate (`learning_rate`):** Set the learning rate to `1e-4` .
- **Batch Size (`batch_size`):** Configure the batch size to `128` .
- **Number of Steps (`n_steps`):** Set the number of steps per update to `2048` .
- **Device (`device`):** Set the device to `"auto"` to utilize available hardware accelerators.

In [9]:
```
import os

# Set rendering backend for MuJoCo environments
# Options: 'egl', 'glfw', 'osmesa'
os.environ['MUJOCO_GL'] = 'egl'  # You can change this if 'egl' causes issue
os.environ['PYOPENGL_PLATFORM'] = 'egl'



# %% Revised PPO Training Cell with Less Logging


# %% Revised PPO Training Cell with Less Logging

############################################################
##### START OF CODE TO BE PRINTED  ##################
############################################################

ENV_ID = "Ant-v5"

env = make_vec_env(ENV_ID, n_envs=4)

ppo_network_args = ppo_network_args = ppo_network_args = dict(
    net_arch=dict(pi=[512, 512], vf=[512, 512]),
    activation_fn=torch.nn.ReLU
)


model = PPO(
    "MlpPolicy",
    env,
    learning_rate=1e-4,
    batch_size=128,
    n_steps=2048,
    policy_kwargs=ppo_network_args,
    device="auto"
)

print("\nNeural Network Architecture:")
print("Actor Network (Policy): Input -> 512 -> ReLU -> 512 -> ReLU -> Output
print("Critic Network (Value): Input -> 512 -> ReLU -> 512 -> ReLU -> Output
```

```
#########################################################
##### END OF CODE TO BE PRINTED ###################
#########################################################



# Training
logger = TrainingLogger(log_interval=200000)
model.learn(total_timesteps=1000000, callback=logger)


# Plot training progress
plt.figure(figsize=(10, 5))
plt.plot(logger.episode_rewards)
plt.title("ppo Training Progress on Ant")
plt.xlabel("Training Epochs")
plt.ylabel("Average Reward")
plt.grid()
plt.show()
```

```
Neural Network Architecture:
Actor Network (Policy): Input -> 512 -> ReLU -> 512 -> ReLU -> Output
Critic Network (Value): Input -> 512 -> ReLU -> 512 -> ReLU -> Output

Step 200000:
  Average reward (last 100 episodes): -90.83
  Current learning rate: 1.00e-04
Step 400000:
  Average reward (last 100 episodes): -55.69
  Current learning rate: 1.00e-04
Step 600000:
  Average reward (last 100 episodes): -12.80
  Current learning rate: 1.00e-04
Step 800000:
  Average reward (last 100 episodes): 21.44
  Current learning rate: 1.00e-04
Step 1000000:
  Average reward (last 100 episodes): 124.12
  Current learning rate: 1.00e-04
```



In [10]:
```
# Evaluate the policy
print(f"Evaluating policy on {ENV_ID}")
# Initialize Gymnasium environment
env = gym.make(ENV_ID, render_mode="rgb_array")

# Evaluate the trained policy
num_eval_episodes = 5
visualize_every = 1  # Visualize every episode
avg_reward = evaluate_policy_sb3(model, env, num_episodes=num_eval_episodes,

env.close()
```

```
Evaluating policy on Ant-v5
Episode 1: Total Reward: 512.13
Episode 2: Total Reward: 1395.60
Episode 3: Total Reward: 580.55
Episode 4: Total Reward: 1365.89
Episode 5: Total Reward: 77.36

**Average Reward over 5 episodes:** 786.31
```

In [ ]:

Behavior Cloning using Minari Dataset

We will be using the Ant dataset from this link: https://minari.farama.org/main/datasets/mujoco/ant/expert-v0/ to train a behavior cloning agent to play Ant-v5.

In [3]:
```
!pip install --user minari
```

```
Collecting minari
  Using cached minari-0.5.2-py3-none-any.whl.metadata (5.8 kB)
Requirement already satisfied: numpy>=1.21.0 in /gpfs/gibbs/project/sds685/s
hared/conda_envs/rl_course/lib/python3.11/site-packages (from minari) (2.2.
2)
Requirement already satisfied: typing-extensions>=4.4.0 in /gpfs/gibbs/proje
ct/sds685/shared/conda_envs/rl_course/lib/python3.11/site-packages (from min
ari) (4.12.2)
Collecting typer>=0.9.0 (from typer[all]>=0.9.0->minari)
  Using cached typer-0.15.1-py3-none-any.whl.metadata (15 kB)
Requirement already satisfied: gymnasium>=0.28.1 in /gpfs/gibbs/project/sds6
85/shared/conda_envs/rl_course/lib/python3.11/site-packages (from minari) (
1.0.0)
Requirement already satisfied: packaging in /gpfs/gibbs/project/sds685/share
d/conda_envs/rl_course/lib/python3.11/site-packages (from minari) (24.2)
Requirement already satisfied: cloudpickle>=1.2.0 in /gpfs/gibbs/project/sds
685/shared/conda_envs/rl_course/lib/python3.11/site-packages (from gymnasium
>=0.28.1->minari) (3.1.1)
Requirement already satisfied: farama-notifications>=0.0.1 in /gpfs/gibbs/pr
oject/sds685/shared/conda_envs/rl_course/lib/python3.11/site-packages (from
gymnasium>=0.28.1->minari) (0.0.4)
Requirement already satisfied: click>=8.0.0 in /gpfs/gibbs/project/sds685/sh
ared/conda_envs/rl_course/lib/python3.11/site-packages (from typer>=0.9.0->t
yper[all]>=0.9.0->minari) (8.1.8)
Collecting shellingham>=1.3.0 (from typer>=0.9.0->typer[all]>=0.9.0->minari)
  Using cached shellingham-1.5.4-py2.py3-none-any.whl.metadata (3.5 kB)
Requirement already satisfied: rich>=10.11.0 in /gpfs/gibbs/project/sds685/s
hared/conda_envs/rl_course/lib/python3.11/site-packages (from typer>=0.9.0->
typer[all]>=0.9.0->minari) (13.9.4)
WARNING: typer 0.15.1 does not provide the extra 'all'
Requirement already satisfied: markdown-it-py>=2.2.0 in /gpfs/gibbs/project/
sds685/shared/conda_envs/rl_course/lib/python3.11/site-packages (from rich>=
10.11.0->typer>=0.9.0->typer[all]>=0.9.0->minari) (3.0.0)
Requirement already satisfied: pygments<3.0.0,>=2.13.0 in /gpfs/gibbs/projec
t/sds685/shared/conda_envs/rl_course/lib/python3.11/site-packages (from rich
>=10.11.0->typer>=0.9.0->typer[all]>=0.9.0->minari) (2.19.1)
Requirement already satisfied: mdurl~=0.1 in /gpfs/gibbs/project/sds685/shar
ed/conda_envs/rl_course/lib/python3.11/site-packages (from markdown-it-py>=
2.2.0->rich>=10.11.0->typer>=0.9.0->typer[all]>=0.9.0->minari) (0.1.2)
Using cached minari-0.5.2-py3-none-any.whl (55 kB)
Using cached typer-0.15.1-py3-none-any.whl (44 kB)
Using cached shellingham-1.5.4-py2.py3-none-any.whl (9.8 kB)
Installing collected packages: shellingham, typer, minari
Successfully installed minari-0.5.2 shellingham-1.5.4 typer-0.15.1
```

In [4]:
```
!pip install --user "minari[hdf5]"
```

```
Collecting minari[hdf5]
  Using cached minari-0.5.2-py3-none-any.whl.metadata (5.8 kB)
Requirement already satisfied: numpy>=1.21.0 in /gpfs/gibbs/project/sds685/s
```

```
hared/conda_envs/rl_course/lib/python3.11/site-packages (from minari[hdf5])
(2.2.2)
Requirement already satisfied: typing-extensions>=4.4.0 in /gpfs/gibbs/proje
ct/sds685/shared/conda_envs/rl_course/lib/python3.11/site-packages (from min
ari[hdf5]) (4.12.2)
Collecting typer>=0.9.0 (from typer[all]>=0.9.0->minari[hdf5])
  Using cached typer-0.15.1-py3-none-any.whl.metadata (15 kB)
Requirement already satisfied: gymnasium>=0.28.1 in /gpfs/gibbs/project/sds6
85/shared/conda_envs/rl_course/lib/python3.11/site-packages (from minari[hdf
5]) (1.0.0)
Requirement already satisfied: packaging in /gpfs/gibbs/project/sds685/share
d/conda_envs/rl_course/lib/python3.11/site-packages (from minari[hdf5]) (24.
2)
Collecting h5py>=3.8.0 (from minari[hdf5])
  Using cached h5py-3.12.1-cp311-cp311-manylinux_2_17_x86_64.manylinux2014_x
86_64.whl.metadata (2.5 kB)
Requirement already satisfied: cloudpickle>=1.2.0 in /gpfs/gibbs/project/sds
685/shared/conda_envs/rl_course/lib/python3.11/site-packages (from gymnasium
>=0.28.1->minari[hdf5]) (3.1.1)
Requirement already satisfied: farama-notifications>=0.0.1 in /gpfs/gibbs/pr
oject/sds685/shared/conda_envs/rl_course/lib/python3.11/site-packages (from
gymnasium>=0.28.1->minari[hdf5]) (0.0.4)
Requirement already satisfied: click>=8.0.0 in /gpfs/gibbs/project/sds685/sh
ared/conda_envs/rl_course/lib/python3.11/site-packages (from typer>=0.9.0->t
yper[all]>=0.9.0->minari[hdf5]) (8.1.8)
Collecting shellingham>=1.3.0 (from typer>=0.9.0->typer[all]>=0.9.0->minari[
hdf5])
  Using cached shellingham-1.5.4-py2.py3-none-any.whl.metadata (3.5 kB)
Requirement already satisfied: rich>=10.11.0 in /gpfs/gibbs/project/sds685/s
hared/conda_envs/rl_course/lib/python3.11/site-packages (from typer>=0.9.0->
typer[all]>=0.9.0->minari[hdf5]) (13.9.4)
WARNING: typer 0.15.1 does not provide the extra 'all'
Requirement already satisfied: markdown-it-py>=2.2.0 in /gpfs/gibbs/project/
sds685/shared/conda_envs/rl_course/lib/python3.11/site-packages (from rich>=
10.11.0->typer>=0.9.0->typer[all]>=0.9.0->minari[hdf5]) (3.0.0)
Requirement already satisfied: pygments<3.0.0,>=2.13.0 in /gpfs/gibbs/projec
t/sds685/shared/conda_envs/rl_course/lib/python3.11/site-packages (from rich
>=10.11.0->typer>=0.9.0->typer[all]>=0.9.0->minari[hdf5]) (2.19.1)
Requirement already satisfied: mdurl~=0.1 in /gpfs/gibbs/project/sds685/shar
ed/conda_envs/rl_course/lib/python3.11/site-packages (from markdown-it-py>=
2.2.0->rich>=10.11.0->typer>=0.9.0->typer[all]>=0.9.0->minari[hdf5]) (0.1.2)
Using cached h5py-3.12.1-cp311-cp311-manylinux_2_17_x86_64.manylinux2014_x86
_64.whl (5.5 MB)
Using cached typer-0.15.1-py3-none-any.whl (44 kB)
Using cached minari-0.5.2-py3-none-any.whl (55 kB)
Using cached shellingham-1.5.4-py2.py3-none-any.whl (9.8 kB)
Installing collected packages: shellingham, h5py, typer, minari
Successfully installed h5py-3.12.1 minari-0.5.2 shellingham-1.5.4 typer-0.1
5.1
```

Let's print out the device we are using. Would be great if we use a GPU

In [5]:
```
import os
import base64
import numpy as np
import matplotlib.pyplot as plt
from pathlib import Path
import gymnasium as gym
from stable_baselines3 import PPO
from stable_baselines3.common.vec_env import VecVideoRecorder, DummyVecEnv
from stable_baselines3.common.env_util import make_vec_env
from stable_baselines3.common.callbacks import BaseCallback


import random
from io import BytesIO
import imageio
import warnings
from IPython.display import Image, display

import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset

import sys
sys.path.append('/home/ark89/.local/lib/python3.11/site-packages')


import minari


warnings.filterwarnings('ignore')

import os

# Set rendering backend for MuJoCo environments
# Options: 'egl', 'glfw', 'osmesa'
os.environ['MUJOCO_GL'] = 'egl'  # You can change this if 'egl' causes issue
os.environ['PYOPENGL_PLATFORM'] = 'egl'


# GPU/CPU detection
device = "cuda" if torch.cuda.is_available() else "cpu"
print("="*50)
print(f"Using device: {device.upper()}")
if device == "cuda":
    print(f"GPU: {torch.cuda.get_device_name(0)}")
```

```
print("="*50)
```

```
==================================================
Using device: CUDA
GPU: Tesla V100-SXM2-32GB
==================================================
```

Helper functions for evaluating the learned policy -- deploy the trained policy on the environment and show the gif

In [6]:
```
def generate_gif(frames, fps=30):
    """
    Generates an animated GIF from a list of frames with infinite looping.

    Args:
        frames (list): List of frames (as NumPy arrays).
        fps (int): Frames per second for the GIF.

    Returns:
        gif_bytes (bytes): The GIF image in bytes.
    """
    with BytesIO() as buffer:
        # 'loop=0' ensures the GIF loops infinitely
        imageio.mimsave(buffer, frames, format='GIF', fps=fps, loop=0)
        gif_bytes = buffer.getvalue()
    return gif_bytes

def display_gif(gif_bytes):
    """
    Displays an animated GIF in the Jupyter notebook with infinite looping.

    Args:
        gif_bytes (bytes): The GIF image in bytes.
    """
    display(Image(data=gif_bytes))

def evaluate_policy(policy, env, num_episodes=10, visualize_every=1, max_ste
    """
    Evaluates a trained policy by running multiple episodes, capturing frame
    generating GIFs for selected episodes, and printing average rewards.

    Args:
        policy (nn.Module): The trained policy network.
        env (gym.Env): The Gymnasium environment.
        num_episodes (int): Number of episodes to evaluate.
        visualize_every (int): Frequency of episodes to visualize (e.g., eve
        max_steps (int): Maximum number of steps per episode.
```

```
    Returns:
        avg_reward (float): The average reward over all episodes.
    """
    # Move policy to CPU and set to evaluation mode
    policy = policy.to("cpu")
    policy.eval()

    returns = []
    frames_to_visualize = []
    all_frames = []  # Initialize all_frames to store frames from all episod

    for episode in range(1, num_episodes + 1):
        frames = []
        state, _ = env.reset(seed=256 * episode)
        done = False
        total_reward = 0
        step = 0

        while not done and step < max_steps:
            # Convert state to tensor on CPU
            state_tensor = torch.tensor(state, dtype=torch.float32).unsqueez
            with torch.no_grad():
                action = policy(state_tensor).squeeze(0).numpy()
            state, reward, done, truncated, _ = env.step(action)
            total_reward += reward

            # Render and collect frame
            frame = env.render()
            if frame is not None:
                frames.append(frame)
            step += 1

        returns.append(total_reward)
        all_frames.append(frames)  # Store frames for this episode
        print(f"Episode {episode}: Total Reward: {total_reward:.2f}")

        # Collect frames for visualization based on visualize_every
        if episode % visualize_every == 0 and frames:
            frames_to_visualize.append(frames)

    avg_reward = np.mean(returns)
    print(f"\n**Average Reward over {num_episodes} episodes:** {avg_reward:.

    # Randomly select one episode's frames to generate and display GIF
    # selected_episode = random.randint(0, num_episodes - 1)
    # selected_frames = all_frames[selected_episode]
    # print(f"\n**Displaying GIF for Randomly Selected Episode {selected_epi
    # gif_bytes = generate_gif(selected_frames)
    # display_gif(gif_bytes)
```

```
    return avg_reward
```

Define Policy Network

In [7]:
```
class PolicyNetwork(nn.Module):
    def __init__(self, state_dim, action_dim, hidden_dim=256):
        super(PolicyNetwork, self).__init__()
        self.net = nn.Sequential(
            nn.Linear(state_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, action_dim)
        )

    def forward(self, x):
        return self.net(x)
```

Load Dataset

In [8]:
```
def load_minari_dataset(dataset_name):
    """
    Loads a dataset using Minari and processes it into states and actions te

    Args:
        dataset_name (str): The name of the Minari dataset to load.

    Returns:
        states (torch.Tensor): Tensor of states.
        actions (torch.Tensor): Tensor of actions.
    """
    dataset = minari.load_dataset(dataset_name, download=True)
    all_observations = []
    all_actions = []
    for episode_data in dataset.iterate_episodes():
        obs = torch.tensor(episode_data.observations, dtype=torch.float32)
        acts = torch.tensor(episode_data.actions, dtype=torch.float32)
        # Ensure observations and actions have the same number of steps
        min_len = min(len(obs), len(acts))
        all_observations.append(obs[:min_len])
        all_actions.append(acts[:min_len])
    states = torch.cat(all_observations)
    actions = torch.cat(all_actions)
    print(f"States shape: {states.shape}, Actions shape: {actions.shape}")
    return states, actions
```

Code that performs training -- This is the function you are going to complete

Objective:
Your task is to complete the `train_behavior_cloning` function, which is designed to train a policy network using **behavior cloning**. Behavior cloning involves training a model to mimic the actions of an expert by learning from a dataset of state-action pairs.

Your Task:
Complete the Missing Section by completing the `train_behavior_cloning` function. You will implement the following steps:

1. **Zero the Gradients:**
    - Before performing a new forward pass, reset the gradients of the optimizer to prevent accumulation from previous iterations.
2. **Forward Pass:**
    - Pass the input `states` through the `policy` network to obtain `predicted_actions`.
3. **Compute Loss:**
    - Calculate the loss between the `predicted_actions` and the actual `actions` using the provided `criterion` (loss function).
4. **Backward Pass:**
    - Perform backpropagation by calling `loss.backward()` to compute the gradients of the loss with respect to the network parameters.
5. **Optimizer Step:**
    - Update the network's weights based on the computed gradients using `optimizer.step()`.

In [9]:
```
#########################################################
##### START OF CODE TO BE PRINTED  ######################
#########################################################


def train_behavior_cloning(policy, dataloader, optimizer, criterion, num_epc
    """
    Trains the policy network using behavior cloning.

    Args:
```

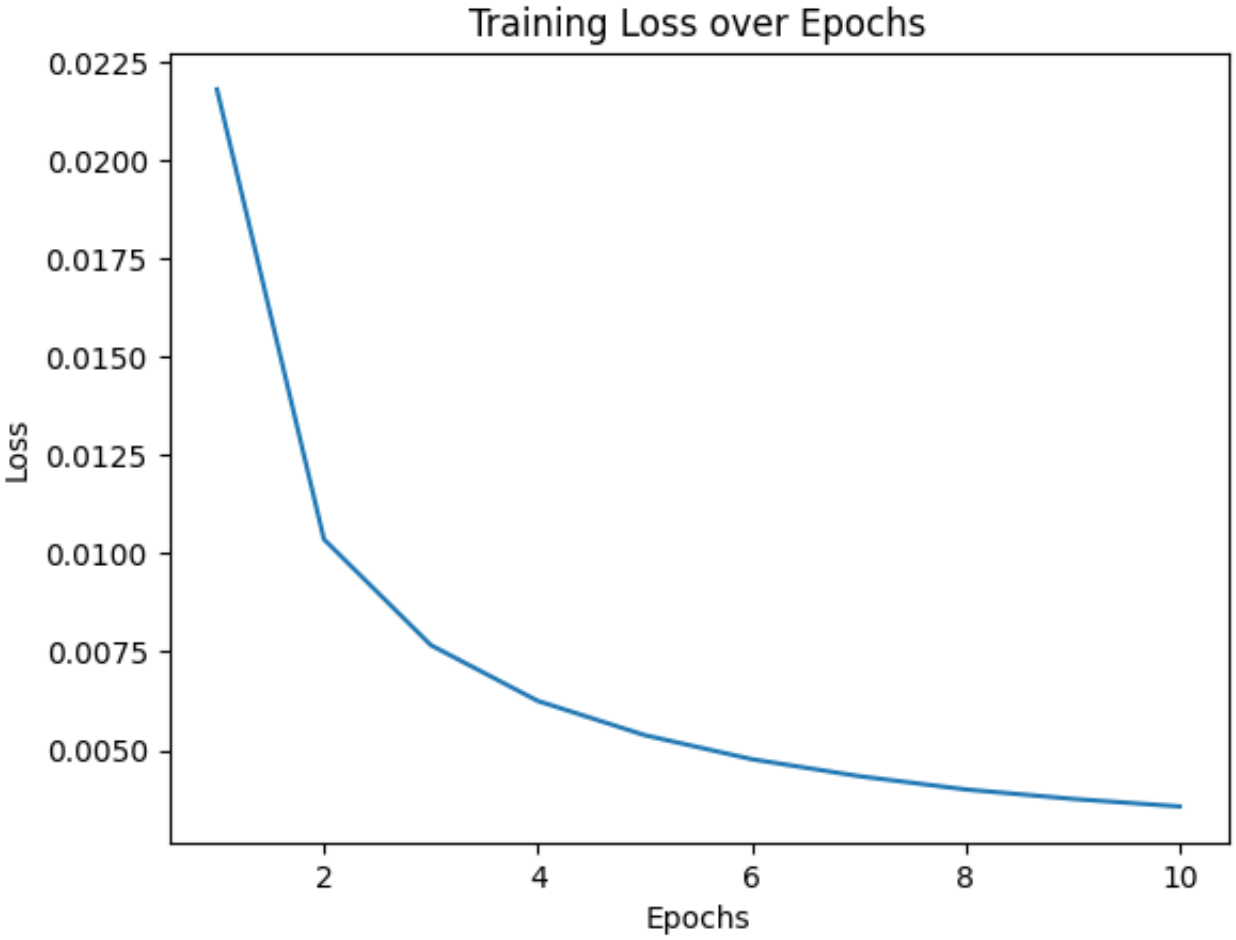
```
        policy (nn.Module): The policy network to train.
        dataloader (DataLoader): DataLoader for training data.
        optimizer (torch.optim.Optimizer): Optimizer for training.
        criterion (nn.Module): Loss function.
        num_epochs (int): Number of training epochs.
        device (str): Device to train on ('cpu' or 'cuda').

    Returns:
        training_losses (list): List of average losses per epoch.
    """
    policy.train()
    training_losses = []
    for epoch in range(num_epochs):
        total_loss = 0
        for states, actions in dataloader:
            states = states.to(device)
            actions = actions.to(device)

            ##########################################################
            ##### COMPLETE CODE ######################
            ##########################################################

            # 1. Zero the gradients from the previous iteration.
            optimizer.zero_grad()

            # 2. Forward pass: compute predicted actions.
            predicted_actions = policy(states)

            # 3. Compute the loss between predicted and actual actions.
            loss = criterion(predicted_actions, actions)

            # 4. Backward pass: compute gradients.
            loss.backward()

            # 5. Optimizer step: update the weights.
            optimizer.step()

            total_loss += loss.item()


        avg_loss = total_loss / len(dataloader)
        training_losses.append(avg_loss)
        if (epoch + 1) % 2 == 0:
            print(f"Epoch {epoch + 1}/{num_epochs}, Loss: {avg_loss:.4f}")
    return training_losses


##########################################################
##### END OF CODE TO BE PRINTED  ######################
##########################################################
```

Training pipeline

In [10]:

```
dataset_name = 'mujoco/ant/expert-v0'  # Replace with your desired dataset r
batch_size = 2048
hidden_dim = 512
learning_rate = 1e-3
num_epochs = 10

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

# Load dataset
states, actions = load_minari_dataset(dataset_name)

# Create dataloader
dataset = TensorDataset(states, actions)

dataloader = DataLoader(
    dataset,
    batch_size=batch_size,
    shuffle=True,
    num_workers=4,  # Use multiple threads for faster data loading
    pin_memory=torch.cuda.is_available()  # Speeds up data transfer to GPU
)


# Initialize policy network, optimizer, and loss function
state_dim = states.shape[1]
action_dim = actions.shape[1]

policy = PolicyNetwork(state_dim, action_dim, hidden_dim).to(device)
optimizer = optim.Adam(policy.parameters(), lr=learning_rate)
criterion = nn.MSELoss()



print("Starting training...")
training_losses = train_behavior_cloning(policy, dataloader, optimizer, crit

# Plot training loss
plt.plot(range(1, num_epochs + 1), training_losses)
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.title("Training Loss over Epochs")
plt.show()
```

```
Using device: cuda
States shape: torch.Size([2000000, 105]), Actions shape: torch.Size([200000
0, 8])
Starting training...
Epoch 2/10, Loss: 0.0104
Epoch 4/10, Loss: 0.0062
Epoch 6/10, Loss: 0.0048
Epoch 8/10, Loss: 0.0040
Epoch 10/10, Loss: 0.0036
```



In [ ]:
```python
# Evaluate the policy
print("Evaluating policy...")
# Initialize Gymnasium environment
env_id = "Ant-v5"  # Replace with your desired environment
env = gym.make(env_id, render_mode="rgb_array")

# Evaluate the trained policy
num_eval_episodes = 5
visualize_every = 1  # Visualize every episode
avg_reward = evaluate_policy(policy, env, num_episodes=num_eval_episodes, vi

env.close()
```

```
Evaluating policy...
Episode 1: Total Reward: 6857.64
Episode 2: Total Reward: 6693.23
Episode 3: Total Reward: 6806.51
Episode 4: Total Reward: 6953.69
```

In [ ]: